31 | 数据流：通过iam-authz-server设计，看数据流服务的设计

2021-08-05 孔令飞

《Go 语言项目开发实战》 课程介绍 >

讲述：孔令飞
时长 19:58 大小 18.29M

你好，我是孔令飞。

在 28 讲 和 29 讲 ，我介绍了 IAM 的控制流服务 iam-apiserver 的设计和实现。这一讲，我们再来看下 IAM 数据流服务 iam-authz-server 的设计和实现。

因为 iam-authz-server 是数据流服务，对性能要求较高，所以采用了一些机制来最大化 API 接口的性能。另外，为了提高开发效率，避免重复造轮子，iam-authz-server 和 iam-apiserver 共享了大部分的功能代码。接下来，我们就来看下，iam-authz-server 是如何跟 iam-apiserver 共享代码的，以及 iam-authz-server 是如何保证 API 接口性能的。

iam-authz-server 的功能介绍

iam-authz-server 目前的唯一功能，是通过提供 `/v1/authz` RESTful API 接口完成资源授权。 `/v1/authz` 接口是通过github.com/ory/ladon来完成资源授权的。

因为 iam-authz-server 承载了数据流的请求，需要确保 API 接口具有较高的性能。为了保证 API 接口的性能，iam-authz-server 在设计上使用了大量的缓存技术。

github.com/ory/ladon 包介绍

因为 iam-authz-server 资源授权是通过 `github.com/ory/ladon` 来完成的，为了让你更好地理解 iam-authz-server 的授权策略，在这里我先介绍下 `github.com/ory/ladon` 包。

Ladon 是用 Go 语言编写的用于实现访问控制策略的库，类似于 RBAC（基于角色的访问控制系统，Role Based Access Control）和 ACL（访问控制列表，Access Control Lists）。但是与 RBAC 和 ACL 相比，Ladon 可以实现更细粒度的访问控制，并且能够在更为复杂的环境中（例如多租户、分布式应用程序和大型组织）工作。

Ladon 解决了这个问题：在特定的条件下，谁能够 / 不能够对哪些资源做哪些操作。为了解决这个问题，Ladon 引入了授权策略。授权策略是一个有语法规范的文档，这个文档描述了谁在什么条件下能够对哪些资源做哪些操作。Ladon 可以用请求的上下文，去匹配设置的授权策略，最终判断出当前授权请求是否通过。下面是一个 Ladon 的授权策略样例：

复制代码

```
{
  "description": "One policy to rule them all.",
  "subjects": ["users:<peter|ken>", "users:maria", "groups:admins"],
  "actions" : ["delete", "<create|update>"],
  "effect": "allow",
  "resources": [
    "resources:articles:<.*>",
    "resources:printer"
  ],
  "conditions": {
    "remoteIP": {
        "type": "CIDRCondition",
        "options": {
            "cidr": "192.168.0.1/16"
        }
    }
  }
}
```

策略（Policy）由若干元素构成，用来描述授权的具体信息，你可以把它们看成一组规则。核心元素包括主题（Subject）、操作（Action）、效力（Effect）、资源（Resource）以及生效条件（Condition）。元素保留字仅支持小写，它们在描述上没有顺序要求。对于没有特定约束条件的策略，Condition 元素是可选项。一条策略包含下面 6 个元素：

主题（Subject），主题名是唯一的，代表一个授权主题。例如，“ken” or “printer-service.mydomain.com”。

操作（Action），描述允许或拒绝的操作。

效力（Effect），描述策略产生的结果是“允许”还是“拒绝”，包括 allow（允许）和 deny（拒绝）。

资源（Resource），描述授权的具体数据。

生效条件（Condition），描述策略生效的约束条件。

描述（Description），策略的描述。

有了授权策略，我们就可以传入请求上下文，由 Ladon 来决定请求是否能通过授权。下面是一个请求示例：

复制代码

```
{
  "subject": "users:peter",
  "action" : "delete",
  "resource": "resources:articles:ladon-introduction",
  "context": {
    "remoteIP": "192.168.0.5"
  }
}
```

可以看到，在 `remoteIP="192.168.0.5"` 生效条件（Condition）下，针对主题（Subject）`users:peter` 对资源（Resource）`resources:articles:ladon-introduction` 的 `delete` 操作（Action），授权策略的效力（Effect）是 `allow` 的。所以 Ladon 会返回如下结果：

```
{
    "allowed": true
}
```

复制代码

Ladon 支持很多 Condition，具体见下表：

极客时间

Condition	描述
CIDRCondition	检查传入的key值是否匹配condition所设定的CIDR，key值是一个remote IP
StringEqualCondition	检查传入的key值是否是字符串类型，并且等于condition所设定的值
BooleanCondition	检查传入的key值是否是bool类型，并且等于condition所设定的值
StringMatchCondition	检查传入的key值是否匹配condition指定的正则规则
EqualsSubjectCondition	检查传入的key值是否匹配subject
StringPairsEqualCondition	检查传入的key值是否是包含两个元素的数组，并且数组中的两个元素是否相等
ResourceContainsCondition	检查传入的key值是否出现在resource字符串中

至于如何使用这些 Condition，你可以参考 Ladon Condition 使用示例。此外，Ladon 还支持自定义 Condition。

另外，Ladon 还支持授权审计，用来记录授权历史。我们可以通过在 ladon.Ladon 中附加一个 ladon.AuditLogger 来实现：

复制代码

```
import "github.com/ory/ladon"
import manager "github.com/ory/ladon/manager/memory"

func main() {
```

```
    warden := ladon.Ladon{
        Manager: manager.NewMemoryManager(),
        AuditLogger: &ladon.AuditLoggerInfo{}
    }

    // ...
}

```

在上面的示例中，我们提供了 ladon.AuditLoggerInfo，该 AuditLogger 会在授权时打印调用的策略到标准错误。AuditLogger 是一个 interface：

复制代码

```
// AuditLogger tracks denied and granted authorizations.
type AuditLogger interface {
    LogRejectedAccessRequest(request *Request, pool Policies, deciders Policie
    LogGrantedAccessRequest(request *Request, pool Policies, deciders Policies
}
```

要实现一个新的 AuditLogger，你只需要实现 AuditLogger 接口就可以了。比如，我们可以实现一个 AuditLogger，将授权日志保存到 Redis 或者 MySQL 中。

Ladon 支持跟踪一些授权指标，比如 deny、allow、not match、error。你可以通过实现 ladon.Metric 接口，来对这些指标进行处理。ladon.Metric 接口定义如下：

复制代码

```
// Metric is used to expose metrics about authz
type Metric interface {
    // RequestDeniedBy is called when we get explicit deny by policy
    RequestDeniedBy(Request, Policy)
    // RequestAllowedBy is called when a matching policy has been found.
    RequestAllowedBy(Request, Policies)
    // RequestNoMatch is called when no policy has matched our request
    RequestNoMatch(Request)
    // RequestProcessingError is called when unexpected error occured
    RequestProcessingError(Request, Policy, error)
}
```

例如，你可以通过下面的示例，将这些指标暴露给 prometheus：

```
type prometheusMetrics struct{}

func (mtr *prometheusMetrics) RequestDeniedBy(r ladon.Request, p ladon.Policy)
func (mtr *prometheusMetrics) RequestAllowedBy(r ladon.Request, policies ladon
func (mtr *prometheusMetrics) RequestNoMatch(r ladon.Request) {}
func (mtr *prometheusMetrics) RequestProcessingError(r ladon.Request, err erro

func main() {

    warden := ladon.Ladon{
        Manager: manager.NewMemoryManager(),
        Metric:  &prometheusMetrics{},
    }

    // ...
}
```

复制代码

在使用 Ladon 的过程中，有两个地方需要你注意：

所有检查都区分大小写，因为主题值可能是区分大小写的 ID。

如果 ladon.Ladon 无法将策略与请求匹配，会默认授权结果为拒绝，并返回错误。

iam-authz-server 使用方法介绍

上面，我介绍了 iam-authz-server 的资源授权功能，这里介绍下如何使用 iam-authz-server，也就是如何调用 `/v1/authz` 接口完成资源授权。你可以通过下面的 3 大步骤，来完成资源授权请求。

第一步，登陆 iam-apiserver，创建授权策略和密钥。

这一步又分为 3 个小步骤。

1. 登陆 iam-apiserver 系统，获取访问令牌：

```
$ token=`curl -s -XPOST -H'Content-Type: application/json' -d'{"username":"adm
```

复制代码

2. 创建授权策略：

复制代码

```
$ curl -s -XPOST -H"Content-Type: application/json" -H"Authorization: Bearer $
```

3. 创建密钥，并从请求结果中提取 secretID 和 secretKey：

复制代码

```
$ curl -s -XPOST -H"Content-Type: application/json" -H"Authorization: Bearer $
{"metadata":{"id":23,"name":"authztest","createdAt":"2021-04-08T07:24:50.07167
```

第二步，生成访问 iam-authz-server 的 token。

iamctl 提供了 `jwt sigin` 子命令，可以根据 secretID 和 secretKey 签发 Token，方便使用。

复制代码

```
$ iamctl jwt sign ZuxvXNfG08BdEMqkTaP41L2DLArlE6Jpqoox 7Sfa5EfAPIwcTLGCfSvqLf0
eyJhbGciOiJIUzI1NiIsImtpZCI6Ilp1eHZYTmZHMDhCZEVNcWtUYVA0MUwyRExBcmxFNkpwcW9veC
```

你可以通过 `iamctl jwt show <token>` 来查看 Token 的内容：

复制代码

```
$ iamctl jwt show eyJhbGciOiJIUzI1NiIsImtpZCI6Ilp1eHZYTmZHMDhCZEVNcWtUYVA0MUwy
Header:
{
    "alg": "HS256",
    "kid": "ZuxvXNfG08BdEMqkTaP41L2DLArlE6Jpqoox",
    "typ": "JWT"
}
Claims:
{
    "aud": "iam.authz.marmotedu.com",
    "exp": 1617845195,
    "iat": 1617837995,
    "iss": "iamctl",
    "nbf": 1617837995
}
```

我们生成的 Token 包含了下面这些信息。

Header

alg：生成签名的算法。

kid：密钥 ID。

typ：Token 的类型，这里是 JWT。

Claims

aud：JWT Token 的接受者。

exp：JWT Token 的过期时间（UNIX 时间格式）。

iat：JWT Token 的签发时间（UNIX 时间格式）。

iss：签发者，因为我们是用 iamctl 工具签发的，所以这里的签发者是 iamctl。

nbf：JWT Token 的生效时间（UNIX 时间格式），默认是签发时间。

第三步，调用`/v1/authz`接口，完成资源授权请求。

请求方法如下：

```
$ curl -s -XPOST -H'Content-Type: application/json' -H'Authorization: Bearer e
{"allowed":true}
```

如果授权通过，会返回：`{"allowed":true}`。 如果授权失败，则返回：

```
{"allowed":false,"denied":true,"reason":"Request was denied by default"}
```

iam-authz-server 的代码实现

接下来，我们来看下 iam-authz-server 的具体实现，我会从配置处理、启动流程、请求处理流程和代码架构 4 个方面来讲解。

iam-authz-server 的配置处理

iam-authz-server 服务的 main 函数位于authzserver.go文件中，你可以跟读代码，了解 iam-authz-server 的代码实现。iam-authz-server 的服务框架设计跟 iam-apiserver 的服务框架设计保持一致，也是有 3 种配置：Options 配置、组件配置和 HTTP 服务配置。

Options 配置见options.go文件：

复制代码

```
type Options struct {
    RPCServer               string
    ClientCA                string
    GenericServerRunOptions *genericoptions.ServerRunOptions
    InsecureServing         *genericoptions.InsecureServingOptions
    SecureServing           *genericoptions.SecureServingOptions
    RedisOptions            *genericoptions.RedisOptions
    FeatureOptions          *genericoptions.FeatureOptions
    Log                     *log.Options
    AnalyticsOptions        *analytics.AnalyticsOptions
}
```

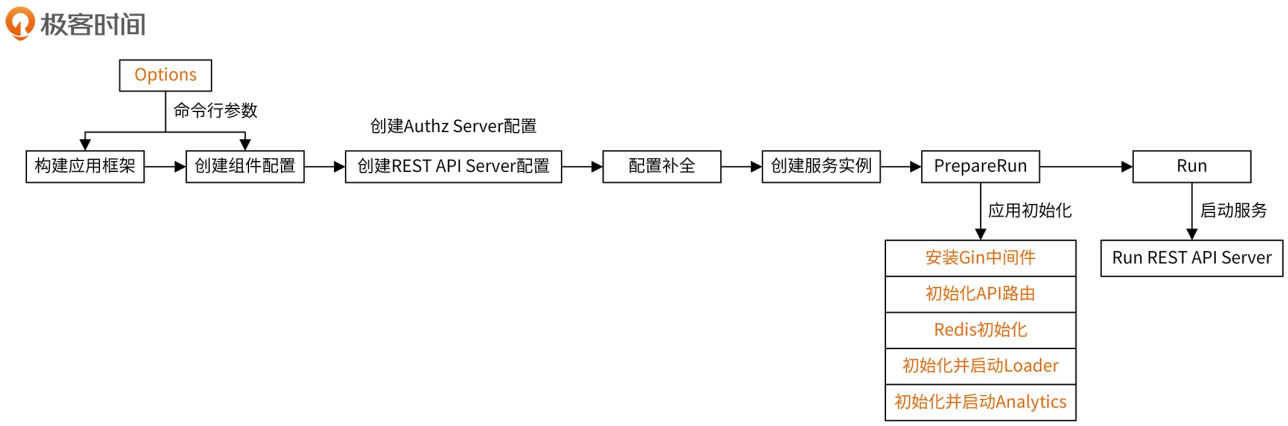
和 iam-apiserver 相比，iam-authz-server 多了 `AnalyticsOptions`，用来配置 iam-authz-server 内的 Analytics 服务，Analytics 服务会将授权日志异步写入到 Redis 中。

iam-apiserver 和 iam-authz-server 共用了 GenericServerRunOptions、InsecureServing、SecureServing、FeatureOptions、RedisOptions、Log 这些配置。所以，我们只需要用简单的几行代码，就可以将很多配置项都引入到 iam-authz-server 的命令行参数中，这也是命令行参数分组带来的好处：批量共享。

iam-authz-server 启动流程设计

接下来，我们来详细看下 iam-authz-server 的启动流程。

iam-authz-server 的启动流程也和 iam-apiserver 基本保持一致。二者比较大的不同在于 Options 参数配置和应用初始化内容。另外，和 iam-apiserver 相比，iam-authz-server 只提供了 REST API 服务。启动流程如下图所示：



iam-authz-server 的 RESTful API 请求处理流程

iam-authz-server 的请求处理流程也是清晰、规范的，具体流程如下图所示：

首先，我们通过 API 调用（`<HTTP Method> + <HTTP Request Path>`）请求 iam-authz-server 提供的 RESTful API 接口 `POST /v1/authz`。

接着，Gin Web 框架接收到 HTTP 请求之后，会通过认证中间件完成请求的认证，iam-authz-server 采用了 Bearer 认证方式。

然后，请求会被我们加载的一系列中间件所处理，例如跨域、RequestID、Dump 等中间件。

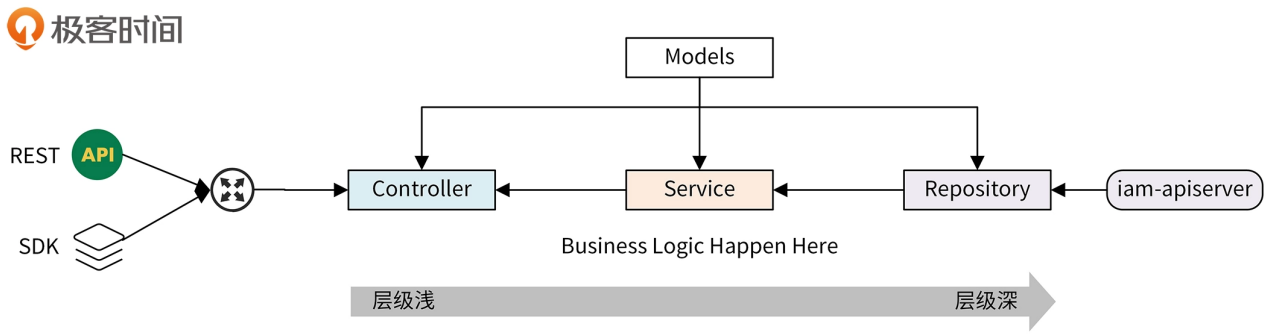
最后，根据<HTTP Method> + <HTTP Request Path>进行路由匹配。

比如，我们请求的 RESTful API 是POST /v1/authz，Gin Web 框架会根据 HTTP Method 和 HTTP Request Path，查找注册的 Controllers，最终匹配到 authzController.Authorize Controller。在 Authorize Controller 中，会先解析请求参数，接着校验请求参数、调用业务层的方法进行资源授权，最后处理业务层的返回结果，返回最终的 HTTP 请求结果。

iam-authz-server 的代码架构

iam-authz-server 的代码设计和 iam-apiserver 一样，遵循简洁架构设计。

iam-authz-server 的代码架构也分为 4 层，分别是模型层（Models）、控制层（Controller）、业务层（Service）和仓库层（Repository）。从控制层、业务层到仓库层，从左到右层级依次加深。模型层独立于其他层，可供其他层引用。如下图所示：



iam-authz-server 和 iam-apiserver 的代码架构有这三点不同：

iam-authz-server 客户端不支持前端和命令行。

iam-authz-server 仓库层对接的是 iam-apiserver 微服务，而非数据库。

iam-authz-server 业务层的代码存放在目录 authorization 中。

iam-authz-server 关键代码分析

和 iam-apiserver 一样，iam-authz-server 也包含了一些优秀的设计思路和关键代码，这里我来一一介绍下。

资源授权

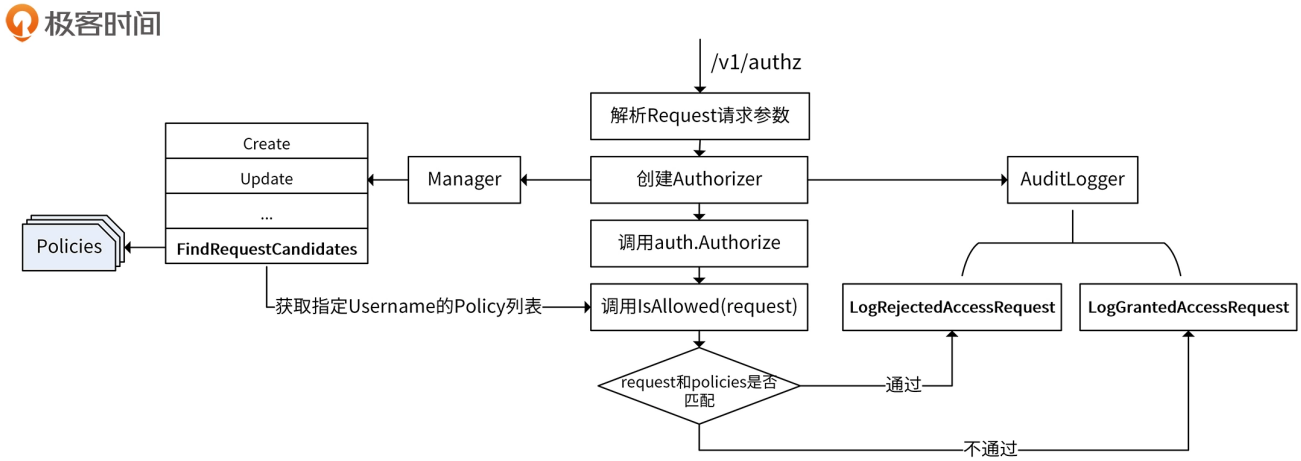
先来看下，iam-authz-server 是如何实现资源授权的。

我们可以调用 iam-authz-server 的 `/v1/authz` API 接口，实现资源的访问授权。`/v1/authz` 对应的 controller 方法是 Authorize：

复制代码

```
func (a *AuthzController) Authorize(c *gin.Context) {
  var r ladon.Request
  if err := c.ShouldBind(&r); err != nil {
    core.WriteResponse(c, errors.WithCode(code.ErrBind, err.Error()), nil)

    return
  }

  auth := authorization.NewAuthorizer(authorizer.NewAuthorization(a.store))
  if r.Context == nil {
    r.Context = ladon.Context{}
  }

  r.Context["username"] = c.GetString("username")
  rsp := auth.Authorize(&r)

  core.WriteResponse(c, nil, rsp)
}
```

该函数使用 `github.com/ory/ladon` 包进行资源访问授权，授权流程如下图所示：



具体分为以下几个步骤：

第一步，在 Authorize 方法中调用 `c.ShouldBind(&r)` ，将 API 请求参数解析到 `ladon.Request` 类型的结构体变量中。

第二步，调用authorization.NewAuthorizer函数，该函数会创建并返回包含 Manager 和 AuditLogger 字段的Authorizer类型的变量。

Manager 包含一些函数，比如 Create、Update 和 FindRequestCandidates 等，用来对授权策略进行增删改查。AuditLogger 包含 LogRejectedAccessRequest 和 LogGrantedAccessRequest 函数，分别用来记录被拒绝的授权请求和被允许的授权请求，将其作为审计数据使用。

第三步，调用auth.Authorize函数，对请求进行访问授权。auth.Authorize 函数内容如下：

复制代码

```
func (a *Authorizer) Authorize(request *ladon.Request) *authzv1.Response {
  log.Debug("authorize request", log.Any("request", request))

  if err := a.warden.IsAllowed(request); err != nil {
    return &authzv1.Response{
      Denied: true,
      Reason: err.Error(),
    }
  }

  return &authzv1.Response{
    Allowed: true,
  }
}
```

该函数会调用 `a.warden.IsAllowed(request)` 完成资源访问授权。IsAllowed 函数会调用 `FindRequestCandidates(r)` 查询所有的策略列表，这里要注意，我们只需要查询请求用户的 policy 列表。在 Authorize 函数中，我们将 username 存入 ladon Request 的 context 中：

复制代码

```
r.Context["username"] = c.GetHeader("username")
```

在FindRequestCandidates函数中，我们可以从 Request 中取出 username，并根据 username 查询缓存中的 policy 列表，FindRequestCandidates 实现如下：

复制代码

```
func (m *PolicyManager) FindRequestCandidates(r *ladon.Request) (ladon.Policie
    username := ""

    if user, ok := r.Context["username"].(string); ok {
      username = user
    }

    policies, err := m.client.List(username)
    if err != nil {
      return nil, errors.Wrap(err, "list policies failed")
    }

    ret := make([]ladon.Policy, 0, len(policies))
    for _, policy := range policies {
      ret = append(ret, policy)
    }

    return ret, nil
  }
```

IsAllowed 函数代码如下：

复制代码

```
func (l *Ladon) IsAllowed(r *Request) (err error) {
    policies, err := l.Manager.FindRequestCandidates(r)
    if err != nil {
        go l.metric().RequestProcessingError(*r, nil, err)
        return err
    }

    return l.DoPoliciesAllow(r, policies)
}
```

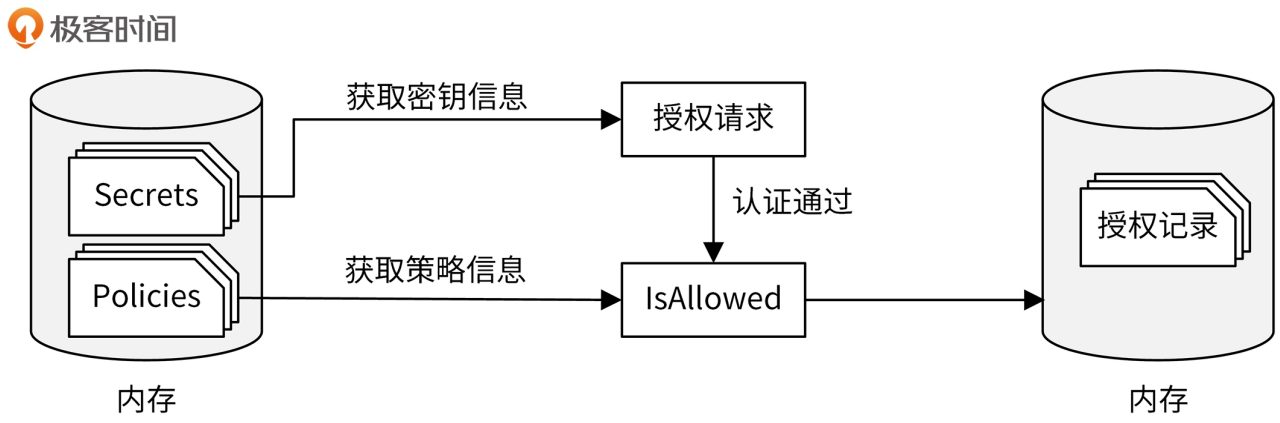
IsAllowed 会调用 DoPoliciesAllow(r, policies) 函数，来匹配 policy 列表。如果 policy 列表中有一条记录跟 request 不匹配，就调用 LogRejectedAccessRequest 函数记录拒绝的请求，并返回值为非 nil 的 error，error 中记录了授权失败的错误信息。

如果所有的 policy 都匹配 request，则调用 LogGrantedAccessRequest 函数记录允许的请求，并返回值为 nil 的 error。

为了降低请求延时，LogRejectedAccessRequest 和 LogGrantedAccessRequest 会将授权记录存储在 Redis 中，之后由 iam-pump 进程读取 Redis，并将授权记录持久化存储在 MongoDB 中。

缓存设计

iam-authz-server 主要用来做资源访问授权，属于数据流的组件，对接口访问性能有比较高的要求，所以该组件采用了缓存的机制。如下图所示：



iam-authz-server 组件通过**缓存密钥和授权策略信息**到内存中，加快密钥和授权策略的查询速度。通过**缓存授权记录**到内存中，提高了授权数据的写入速度，从而大大降低了授权请求接口的延时。

上面的缓存机制用到了 Redis key-value 存储，所以在 iam-authz-server 初始化阶段，需要先建立 Redis 连接（位于initialize函数中）：

```
1 go storage.ConnectToRedis(ctx, s.buildStorageConfig())
```

复制代码

这个代码会维护一个 Redis 连接，如果 Redis 连接断掉，会尝试重连。这种方式可以使我们在调用 Redis 接口进行数据读写时，不用考虑连接断开的问题。

接下来，我们就来详细看看，iam-authz-server 是如何实现缓存机制的。

先来看下密钥和策略缓存。

iam-authz-server 通过load包来完成密钥和策略的缓存。

在 iam-authz-server 进程启动时，会创建并启动一个 Load 服务（位于initialize函数中）：

复制代码

```
load.NewLoader(ctx, cacheIns).Start()
```

先来看创建 Load 服务。创建 Load 服务时，传入了 cacheIns 参数，cacheIns 是一个实现了Loader接口的实例：

复制代码

```
type Loader interface {
    Reload() error
}
```

然后看启动 Load 服务。通过 Load 实例的 Start 方法来启动 Load 服务：

复制代码

```
func (l *Load) Start() {
    go startPubSubLoop()
    go l.reloadQueueLoop()
    go l.reloadLoop()

    l.DoReload()
}
```

Start 函数先启动了 3 个协程，再调用 `l.DoReload()` 完成一次密钥和策略的同步：

复制代码

```
func (l *Load) DoReload() {
    l.lock.Lock()
```

```
    defer l.lock.Unlock()

    if err := l.loader.Reload(); err != nil {
        log.Errorf("faild to refresh target storage: %s", err.Error())
    }

    log.Debug("refresh target storage succ")
}
```

上面我们说了，创建 Load 服务时，传入的 cacheIns 实例是一个实现了 Loader 接口的实例，所以在DoReload方法中，可以直接调用 Reload 方法。cacheIns 的 Reload 方法会从 iam-apiserver 中同步密钥和策略信息到 iam-authz-server 缓存中。

我们再来看下，startPubSubLoop、reloadQueueLoop、reloadLoop 这 3 个 Go 协程分别完成了什么功能。

1. startPubSubLoop 协程

startPubSubLoop函数通过StartPubSubHandler函数，订阅 Redis 的 `iam.cluster.notifications` channel，并注册一个回调函数：

复制代码

```
func(v interface{}) {
    handleRedisEvent(v, nil, nil)
}
```

handleRedisEvent函数中，会将消息解析为Notification类型的消息，并判断 Command 的值。如果是 NoticePolicyChanged 或 NoticeSecretChanged，就会向 `reloadQueue` channel 中写入一个回调函数。因为我们不需要用回调函数做任何事情，所以这里回调函数是 nil。 `reloadQueue` 主要用来告诉程序，需要完成一次密钥和策略的同步。

2. reloadQueueLoop 协程

reloadQueueLoop 函数会监听 `reloadQueue` ，当发现有新的消息（这里是回调函数）写入时，会实时将消息缓存到 `requeue` 切片中，代码如下：

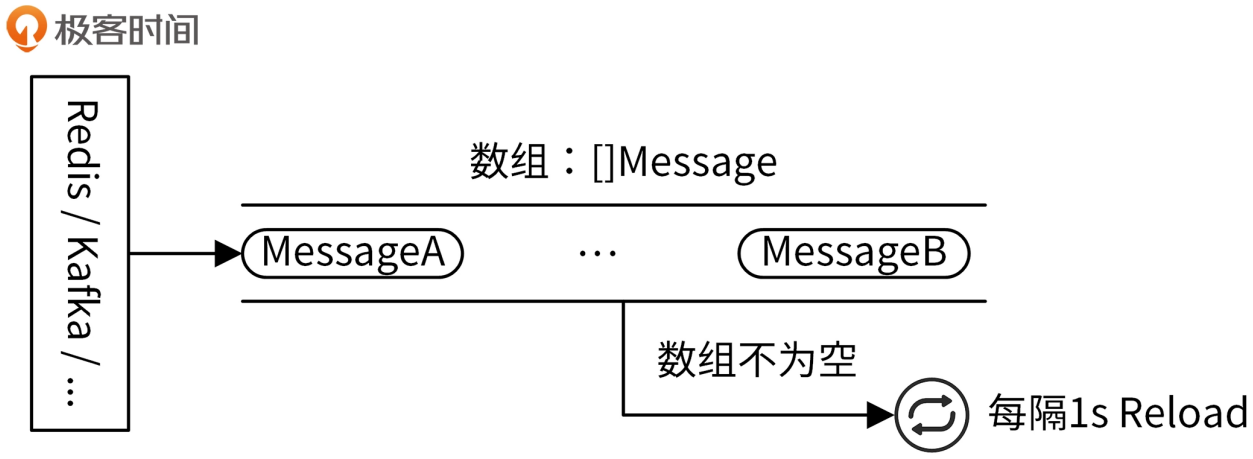
复制代码

```
func (l *Load) reloadQueueLoop(cb ...func()) {
    for {
      select {
      case <-l.ctx.Done():
        return
      case fn := <-reloadQueue:
        requeueLock.Lock()
        requeue = append(requeue, fn)
        requeueLock.Unlock()
        log.Info("Reload queued")
        if len(cb) != 0 {
          cb[0]()
        }
      }
    }
  }
```

3. reloadLoop 协程

通过reloadLoop函数启动一个 timer 定时器，每隔 1 秒会检查 `requeue` 切片是否为空，如果不为空，则调用 `l.DoReload` 方法，从 iam-apiserver 中拉取密钥和策略，并缓存在内存中。

密钥和策略的缓存模型如下图所示：



密钥和策略缓存的具体流程如下：

接收上游消息（这里是从 Redis 中接收），将消息缓存到切片或者带缓冲的 channel 中，并启动一个消费协程去消费这些消息。这里的消费协程是 reloadLoop，reloadLoop 会每隔 1s 判断 `requeue` 切片是否长度为 0，如果不为 0，则执行 `l.DoReload()` 缓存密钥和策略。

讲完了密钥和策略缓存，**再来看下授权日志缓存。**

在启动 iam-authz-server 时，还会启动一个 Analytics 服务，代码如下（位于 internal/authzserver/authzserver.go文件中）：

复制代码

```
    if s.analyticsOptions.Enable {
        analyticsStore := storage.RedisCluster{KeyPrefix: RedisKeyPrefix}
        analyticsIns := analytics.NewAnalytics(s.analyticsOptions, &analyticsS
        analyticsIns.Start()
        s.gs.AddShutdownCallback(shutdown.ShutdownFunc(func(string) error {
            analyticsIns.Stop()

            return nil
        }))
    }
```

NewAnalytics函数会根据配置，创建一个 Analytics 实例：

复制代码

```
func NewAnalytics(options *AnalyticsOptions, store storage.AnalyticsHandler) *
    ps := options.PoolSize
    recordsBufferSize := options.RecordsBufferSize
    workerBufferSize := recordsBufferSize / uint64(ps)
    log.Debug("Analytics pool worker buffer size", log.Uint64("workerBufferSiz

    recordsChan := make(chan *AnalyticsRecord, recordsBufferSize)

    return &Analytics{
      store:                    store,
      poolSize:                 ps,
      recordsChan:              recordsChan,
      workerBufferSize:         workerBufferSize,
      recordsBufferFlushInterval: options.FlushInterval,
    }
  }
```

上面的代码创建了一个带缓冲的 recordsChan ：

复制代码

```
recordsChan := make(chan *AnalyticsRecord, recordsBufferSize)
```

recordsChan 存放的数据类型为AnalyticsRecord，缓冲区的大小为 recordsBufferSize（通过 --analytics.records-buffer-size 选项指定）。可以通过RecordHit函数，向recordsChan 中写入 AnalyticsRecord 类型的数据：

复制代码

```
func (r *Analytics) RecordHit(record *AnalyticsRecord) error {
    // check if we should stop sending records 1st
    if atomic.LoadUint32(&r.shouldStop) > 0 {
        return nil
    }

    // just send record to channel consumed by pool of workers
    // leave all data crunching and Redis I/O work for pool workers
    r.recordsChan <- record

    return nil
}
```

iam-authz-server 是通过调用 LogGrantedAccessRequest 和 LogRejectedAccessRequest 函数来记录授权日志的。在记录授权日志时，会将授权日志写入 recordsChan channel 中。LogGrantedAccessRequest函数代码如下：

复制代码

```
func (auth *Authorization) LogGrantedAccessRequest(r *ladon.Request, p ladon.P
    conclusion := fmt.Sprintf("policies %s allow access", joinPoliciesNames(d)
    rstring, pstring, dstring := convertToString(r, p, d)
    record := analytics.AnalyticsRecord{
        TimeStamp:  time.Now().Unix(),
        Username:   r.Context["username"].(string),
        Effect:     ladon.AllowAccess,
        Conclusion: conclusion,
        Request:    rstring,
        Policies:   pstring,
        Deciders:   dstring,
    }

```

```
    record.SetExpiry(0)
    _ = analytics.GetAnalytics().RecordHit(&record)
}
```

上面的代码，会创建 AnalyticsRecord 类型的结构体变量，并调用 RecordHit 将变量的值写入 recordsChan channel 中。将授权日志写入 recordsChan channel 中，而不是直接写入 Redis 中，这可以大大减少写入延时，减少接口的响应延时。

还有一个 worker 进程从 recordsChan 中读取数据，并在数据达到一定阈值之后，批量写入 Redis 中。在Start函数中，我们创建了一批 worker，worker 个数可以通过 --analytics.pool-size 来指定 。Start 函数内容如下：

复制代码

```
func (r *Analytics) Start() {
    analytics = r
    r.store.Connect()

    // start worker pool
    atomic.SwapUint32(&r.shouldStop, 0)
    for i := 0; i < r.poolSize; i++ {
      r.poolWg.Add(1)
      go r.recordWorker()
    }

    // stop analytics workers
    go r.Stop()
  }
```

上面的代码通过 go r.recordWorker() 创建了 由poolSize 指定个数的recordWorker（worker），recordWorker 函数会从 recordsChan 中读取授权日志并存入 recordsBuffer 中，recordsBuffer 的大小为 workerBufferSize，workerBufferSize 计算公式为：

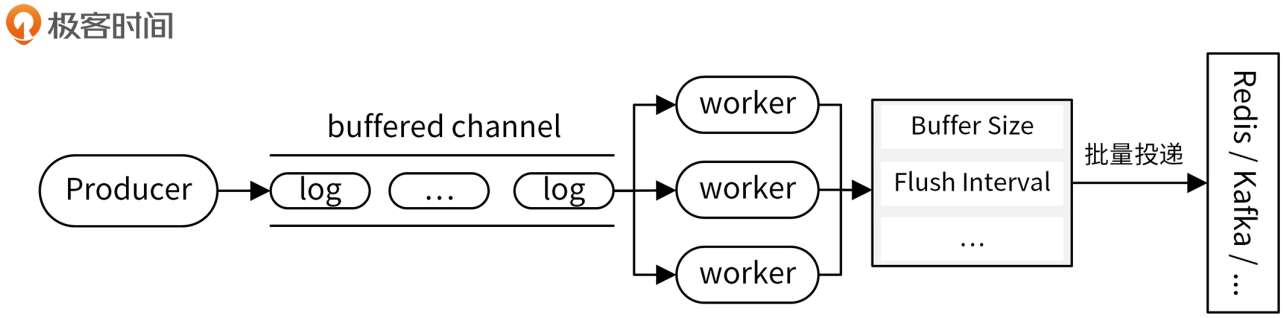
复制代码

```
ps := options.PoolSize
recordsBufferSize := options.RecordsBufferSize
workerBufferSize := recordsBufferSize / uint64(ps)
```

其中，options.PoolSize 由命令行参数 `--analytics.pool-size` 指定，代表 worker 的个数，默认 50；options.RecordsBufferSize 由命令行参数 `--analytics.records-buffer-size` 指定，代表缓存的授权日志消息数。也就是说，我们把缓存的记录平均分配给所有的 worker。

当 recordsBuffer 存满或者达到投递最大时间后，调用 `r.Store.AppendToSetPipelined(analyticsKeyName, recordsBuffer)` 将记录批量发送给 Redis，为了提高传输速率，这里将日志内容编码为 msgpack 格式后再传输。

上面的缓存方法可以抽象成一个缓存模型，满足实际开发中的大部分需要异步转存的场景，如下图所示：

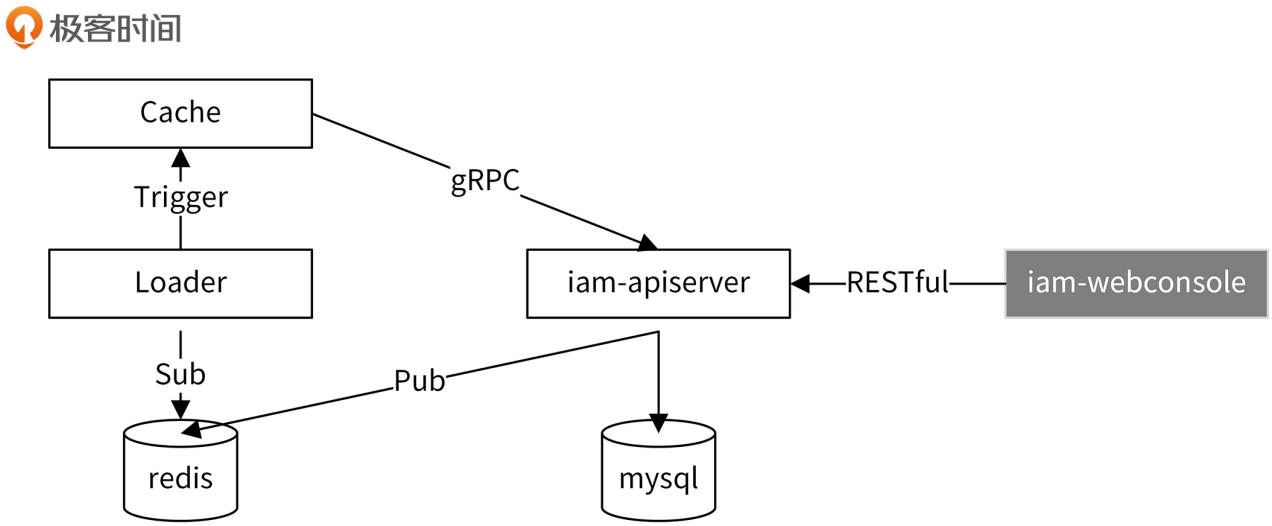


Producer 将数据投递到带缓冲的 channel 中，后端有多个 worker 消费 channel 中的数据，并进行批量投递。你可以设置批量投递的条件，一般至少包含**最大投递日志数**和**最大投递时间间隔**这两个。

通过以上缓冲模型，你可以将日志转存的时延降到最低。

数据一致性

上面介绍了 iam-authz-server 的 `/v1/authz` 接口，为了最大化地提高性能，采用了大量的缓存设计。因为数据会分别在持久化存储和内存中都存储一份，就可能会出现数据不一致的情况。所以，我们也要确保缓存中的数据和数据库中的数据是一致的。数据一致性架构如下图所示：



密钥和策略同步流程如下：

1. 通过 iam-webconsole 请求 iam-apiserver 创建（或更新、删除）密钥（或策略）。
2. iam-apiserver 收到“写”请求后，会向 Redis `iam.cluster.notifications` channel 发送 PolicyChanged 或 SecretChanged 消息。
3. Loader 收到消息后，会触发 cache loader 实例执行 `Reload` 方法，重新从 iam-apiserver 中同步密钥和策略信息。

Loader 不会关心 `Reload` 方法的具体实现，只会在收到指定消息时，执行 `Reload` 方法。通过这种方式，我们可以实现不同的缓存策略。

在 cache 实例的 `Reload` 方法中，我们其实是调用仓库层 Secret 和 Policy 的 List 方法来获取密钥和策略列表。仓库层又是通过执行 gRPC 请求，从 iam-apiserver 中获取密钥和策略列表。

cache 的Reload方法，会将获取到的密钥和策略列表缓存在ristretto类型的 Cache 中，供业务层调用。业务层代码位于internal/authzserver/authorization目录下。

总结

这一讲中，我介绍了 IAM 数据流服务 iam-authz-server 的设计和实现。iam-authz-server 提供了 `/v1/authz` RESTful API 接口，供第三方用户完成资源授权功能，具体是

使用 Ladon 包来完成资源授权的。Ladon 包解决了“在特定的条件下，谁能够 / 不能够对哪些资源做哪些操作”的问题。

iam-authz-server 的配置处理、启动流程和请求处理流程跟 iam-apiserver 保持一致。此外，iam-authz-server 也实现了简洁架构。

iam-authz-server 通过缓存密钥和策略信息、缓存授权日志来提高 `/v1/authz` 接口的性能。

在缓存密钥和策略信息时，为了和 iam-apiserver 中的密钥和策略信息保持一致，使用了 Redis Pub/Sub 机制。当 iam-apiserver 有密钥 / 策略变更时，会往指定的 Redis channel Pub 一条消息。iam-authz-server 订阅相同的 channel，在收到新消息时，会解析消息，并重新从 iam-apiserver 中获取密钥和策略信息，缓存在内存中。

iam-authz-server 执行完资源授权之后，会将授权日志存放在一个带缓冲的 channel 中。后端有多个 worker 消费 channel 中的数据，并进行批量投递。可以设置批量投递的条件，例如最大投递日志数和最大投递时间间隔。

课后练习

1. iam-authz-server 和 iam-apiserver 共用了应用框架（包括一些配置项）和 HTTP 服务框架层的代码，请阅读 iam-authz-server 代码，看下 IAM 项目是如何实现代码复用的。
2. iam-authz-server 使用了 ristretto 来缓存密钥和策略信息，请调研下业界还有哪些优秀的缓存包可供使用，欢迎在留言区分享。

欢迎你在留言区与我交流讨论，我们下一讲见。

分享给需要的人，Ta订阅后你可得 24 元现金奖励

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | ORM：CURD 神器 GORM 包介绍及实战

下一篇 32 | 数据处理：如何高效处理应用程序产生的数据？

更多课程推荐

精选留言 (2)

写留言

daz2yy
2021-08-06

老师，请问下，数据流和控制流这个怎么来理解呢，是从什么角度来定义的服务类型的？

作者回复: 数据流承载核心业务功能。

控制流一般用来创建一些资源，供数据流使用。这些，资源一旦建成，控制流可以一直使用。所以这时候控制流挂了，资源还是存在的，数据流仍然能够正常运转，也就是不影响业务功能。

hello
2021-08-05

老师，咨询您一个问题，使用GiN做WEB服务，微服务间通过gRPC通讯，如何选择配置注册中心，老师能否推荐几款比较流行开源的配置注册中心。

展开

作者回复: 网关可以使用，tyk：https://github.com/TykTechnologies/tyk。

tyk后端开源，前端闭源。你可以自己开发一套前端。

配置中心可以试试：https://github.com/apache/servicecomb-service-center

servicecomb-service-center是华为开源的注册中心，基于etcd封装。该注册中心可以无缝对接华为开源的微服务框架go-chassis